# *Java Packaging. Containers* (1)

- Canonical Java Packaging
  - JAR Files. Fat JARs
  - JPMS + `jlink`
  - GraalVM `native-image`
- Linux Packaging (DEB, RPM)
- Virtualization
- Containers: Concepts, Tools, Implementation
- Container Orchestration
- Some Container Solutions

# Deadline-Driven Development™

Apr **23**: Second Release

- All previously found **defects fixed**
- **At least 2** User Stories implemented
- Proper **error handling**
- **Unit Tests** for all classes except framework-related
  - Line/Statement **Coverage ≥ 70%**

# Canonical Java Packaging: JAR

- Just fancy **ZIP archives** containing **compiled Java classes**, **resources** and **metainformation** (`META-INF/`)

  *@see* https://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html
- **Compiled classes** and **class resources** are put into directories corresponding to Java packages.
  *E.g.* class `ru.hse.java.HelloWorld => ru/hse/java/HelloWorld.class`
  - No directories are created for anonymous, inner and static inner classes, because they are synthesized by the compiler
    *E.g.* class `ru.hse.java.HelloWorld.MyCoolClass => ru/hse/HelloWorld$MyCoolClass.class`
- Most important **metainformation** is the Manifest, `META-INF/MANIFEST.MF`:

```
Manifest-Version: 1.0
Main-Class: ru.hse.MyCoolProjectMain
<more key-value pairs...>
```

→ `java -jar my-cool-project-1.0.jar`

- `META-INF/` directory also MAY contain:
  - Digital signature files (`*.RSA, *.DSA, SIG-*`)
  - **Service Provider** definitions (`META-INF/services/<FQCN of Service Class Impl>`). *@see* future seminar on DI

# Fat (Uber) JAR

https://stackoverflow.com/a/36539885/3438672

- Bring all of your dependencies (transitively) in a single large JAR file
- Three methods:
  - **Unpack all dependency JARs (`maven-assembly-plugin` → `jar-with-dependencies`)**
  - Unpack all dependency JARs, but also rename their packages and merge resources (`maven-shade-plugin`)
    - To avoid conflicts with your library users' dependencies
  - Copy JARs into your JAR and use a special *class loader* to transparently access their classes (`onejar`, `spring-boot-plugin:repackage`)
- Classpath is Linear → JAR Hell
  - `maven-enforcer-plugin` helps, use it!

# JPMS + jlink

https://www.baeldung.com/java-9-modularity

- **Java Platform Module System**
  - Modules depend on each other (**`requires [transitive] ...`**)
  - Modules have Strong Encapsulation™
    - Explicitly define which packages are visible to the outside world (**`exports ... [to ...]`**)
    - Explicitly define which packages they allow reflection access to (**`opens ... [to ...]`**)
  - Dependency Cycles and Split Packages disallowed
  - Inter-module dependencies are assumed to be *mostly* static (determined at app start time)
- Requires you to **cleanly separate your system into modules**, which is **NO SMALL TASK**
- But as a bonus, you can package only **modules really used** by your application:

```
jlink --module-path $JAVA_HOME/jmods:mlib \
      --add-modules com.greetings \
      --output greetingsapp
```

...the image will include **only used** JDK modules (**jmods:…**)

# GraalVM `native-image`

- Ahead-of-Time compiles your Java code into a **native application for the target platform**
  - Some limitations, mostly around Reflection usage and static class initialization (**`static final`**s and **`static`** `{}` blocks spawning threads and the like)
  - Popular frameworks are supported out-of-box, less popular can encounter problems
- Emerging frameworks using native-image as the default: Quarkus, Micronaut
- Good fit for Microservices, Serverless, CLIs, Agents, …
  - **Fast start** (low start latency) far more important than **peak throughput**
  - GC and runtime is naive compared to OpenJDK's Hotspot

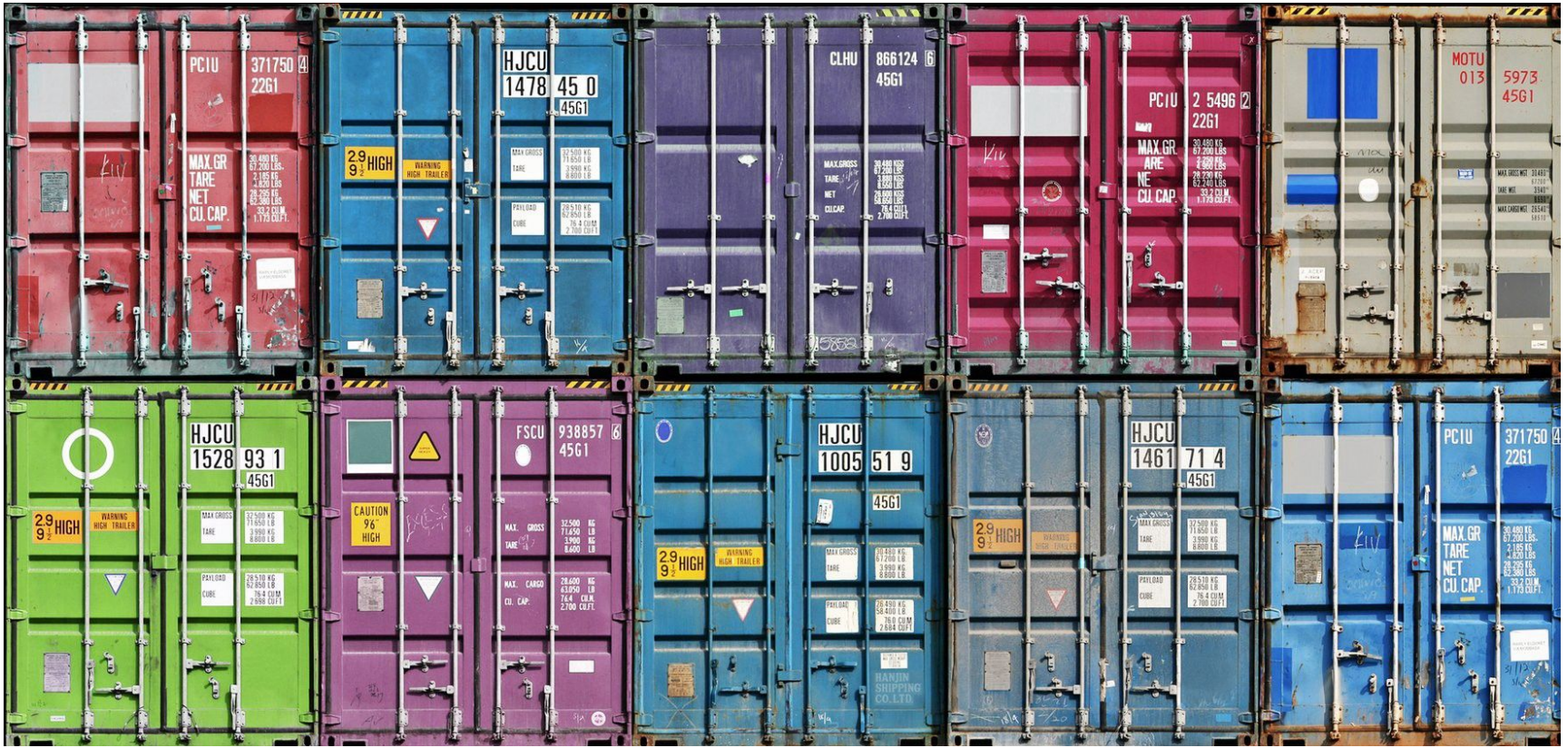
# Linux Packaging (RPM, DEB)

- General tools for managing components of a Linux distribution
  - Manage *packages*, which provide *files* installed into your filesystem
  - Packages are *versioned* and *depend* on each other (possibly with a version/version range specified)
  - Package Managers provide dependency resolution, download and installation
- Cannot have multiple versions of the same package
  - Multiple packages, however, can provide the same binary, *e.g.* `/bin/java` via `update-alternatives`

# Virtualization

- Run 1+ *Guests* (OS w/virtualized hardware) on a single *Host*, managed by a *Hypervisor*
  - *Reasonably fast* now*:* hardware (*e.g.,* Intel VT-x) and software-assisted (*Para*virtualization)
- Advantages
  - **Tight Isolation** (CPU and memory access, virtualized network, virtualized storage, …)
  - **Better Resource Utilization:** A single host server can manage multiple guest VMs
  - **Better Scalability:** both Vertical (allocate more host resources to guest) and Horizontal (spawn more guests)
  - **Better Disaster Recovery**
    - VM crashed? Just spawn a new one instead
    - VMs have transparent access to replicated network storage
- Drawbacks
  - **You bring *Everything but the kitchen sink*:** Full OS image + all packages + your app (**orders of magnitude** smaller)
  - **Expensive VM setup and teardown** (tens of seconds..minutes, depending on your workload and resources available)
  - ***Some* performance degradation** (`top` → Steal time)

# Containers! Containers! Containers!



# Containers: Concept

- Hypervisors virtualize hardware and then run a full-fledged real OS on it
- But modern OSs **already have powerful primitives** for process, filesystem, network isolation and resource management!

**Container** includes all relevant (software and data, **NOT** OS!) dependencies for your application/service

…unlike VMs, you cannot *e.g.* run on a fixed specific kernel version. But this is rarely required for most common apps and services anyway

# Containers: Packaging & Delivery

- Linux Containers were intended for **standard packaging and delivery** of software
  - To [re]create development environment
  - To run multiple different versions concurrently
- It is in contrast to *e.g.* FreeBSD Jails, which were designed a security feature first and foremost
- Docker Images, OCI
  - Optimized for efficient downloads and caching:
    - Layered filesystem
    - Hash-based layer identity
  - Images are tagged (typically with version). `latest` tag (convention)
- Container Registries (similar to Maven artifact repositories)
  - DockerHub
  - Private installations
  - Cloud offerings from all major cloud providers, incl. Yandex.Cloud

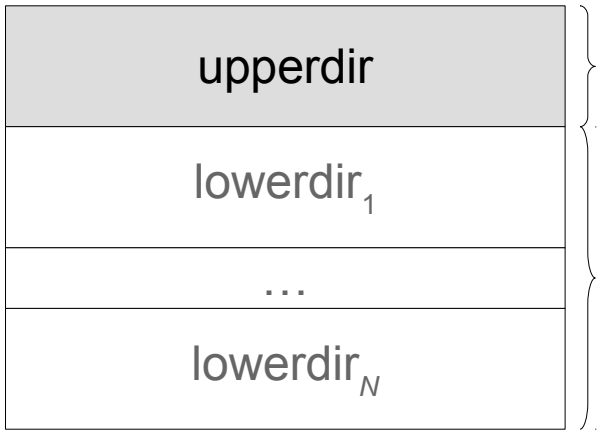
# Containers: Isolation → Namespaces

**Unit of isolation** in Containers is the **Process**

- Namespaces limit **visibility** of sensitive system entities: processes, network interfaces, users, mount points, …
- **Process isolation:** main process in container runs as PID 1, but its PID in the host is different
- **Network isolation:** container has limited access to host network interfaces, can have special *bridge* network interfaces etc.
- **User isolation:** process inside container runs as root (or as some user that you specify)
- **Mount point isolation:** containers cannot access host storage unless explicitly specified

# Containers: Isolation → FS

- `pivot_root` (`chroot` on steroids) to have `/` independent of host root filesystem
- Union Filesystem, *overlayfs*



# Containers: Limits and Security

- cgroups (Control Groups): Hierarchical Resource Accounting and Limits
  - Limit CPU core usage
  - Memory usage (RSS, resident set size)
  - I/O (read and write iops)
  - Network bandwidth
  - Process is typically killed or throttled when resource overuse is detected
- Container Security: Principle of least privilege
  - Drop capabilities (`CAP_...`)
  - seccomp-bpf: Selective filtering of syscalls

# Container Orchestration

- Containers are much more lightweight that VMs
  - You can have tens and even hundreds of them running on a modest VM!
- Container Orchestration (*E.g.* Kubernetes, Docker Swarm, Nomad, Amazon ECS, …): Managing lots of containers, and Clusters of similar containers
  - Resource Allocator
  - Workload Scheduler
    - Jobs: batch, throughput-oriented workloads
    - Services (stateless and stateful): interactive, latency-oriented workloads
  - Resilience: restarts/retries, container migration
  - Autoscaling
  - Persistent storage management
- Container Management Philosophy: Cattle **NOT** Pets!

# Some Container Solutions

- **Basic Container Management: *Docker*** (safer, simpler production alternative is *rkt*)
  - build image, push (publish to registry), pull (download from registry)
  - run image/run command inside image
  - view stdout/stderr of container process/route process logs to syslog and whatever
- **Basic Orchestration: *Docker Compose***
- Advanced Orchestration: *Hashicorp Nomad*, *Kubernetes*
- Container-Optimized Linux distributions (*Alpine*, *distroless*):
  Minimal dependencies required to run containers →
  → Smaller attack surface and better performance
- Container Operating Systems: CoreOS, AWS Bottlerocket (experimental)